Centro Universitário SENAC
Engenharia de Software II

Trabalho Código Limpo

Bryan Laquimam Lubke Gonçalves
Bernardo Ginar Carvalho
Filipe Silveira Maciel
Pedro Hasse Niemczewski

Código a ser analisado:

```
class AtendimentoRepository implements IAtendimentoRepository {
  private readonly PAGE_SIZE = 20;

  public async getAtendimentos(
    page: number = 1,
    statusId?: string,
    nome?: string,
    numeroInternacao?: string,
    setor?: string,
    leito?: string,
    convenioId?: string,
    dataEntrada?: string,
    year?: number,
    month?: number
  ): Promise<IAtendimentoDto[]> {
    try {
      const skip = (page - 1) * this.PAGE_SIZE;

      let filter: any = {};

      if (statusId) {
        filter.statusId = statusId;
      }
      if (nome) {
        filter.nome = { contains: nome };
      }
      if (numeroInternacao) {
        filter.numeroInternacao = { contains: numeroInternacao };
      }
      if (setor) {
        filter.setor = setor;
      }
      if (leito) {
        filter.leito = leito;
      }
      if (convenioId) {
        filter.convenioId = convenioId;
      }
      if (dataEntrada) {
        filter.dataEntrada = dataEntrada;
      }

      if (year) {
        filter.dataEntrada = {
          gte: new Date(year, 0, 1),
          lt: new Date(Number(year) + 1, 0, 1),
        };
      }

      if (month) {
```

```
      filter.mesEntrada = {
        equals: month-1,
      };
    }

    const atendimentos = await prisma.atendimento.findMany({
      skip: skip,
      take: this.PAGE_SIZE,
      orderBy: {
        createdAt: "desc",
      },
      where: filter,
    });

    let dtosAtendimento: IAtendimentoDto[] = [];
    for (const atendimento of atendimentos) {
      dtosAtendimento.push(
        new AtendimentoDto(
          atendimento.id.toString(),
          atendimento.numeroInternacao,
          atendimento.nome,
          atendimento.statusId,
          atendimento.tipoAtendimentoId,
          atendimento.quantidadeFr,
          atendimento.quantidadeFm,
          atendimento.setor,
          atendimento.leito,
          atendimento.convenioId,
          atendimento.dataEntrada,
          atendimento.dataSaida,
          atendimento.mesEntrada
        )
      );
    }

    return dtosAtendimento;
  } catch (error: any) {
    throw new Error(`Desculpe, um erro no servidor ocorreu! Erro: ${error}`);
  }
}

public async getAtendimentosBySetor(setor: Setor, page: number = 1): Promise<IAtendimentoDto[]> {
  try {
    const skip = (page - 1) * this.PAGE_SIZE;

    let dtosAtendimento: IAtendimentoDto[] = [];

    const setorEnum = setor.toUpperCase() as keyof typeof Setor;

    const atendimentos = await prisma.atendimento.findMany({
      where: {
        setor: {
```

```
        equals: setorEnum,
      },
    },
    skip: skip,
    take: this.PAGE_SIZE,
    orderBy: {
      createdAt: "desc",
    },
  });

  for (const atendimento of atendimentos) {
    dtosAtendimento.push(
      new AtendimentoDto(
        atendimento.id.toString(),
        atendimento.numeroInternacao,
        atendimento.nome,
        atendimento.statusId,
        atendimento.tipoAtendimentoId,
        atendimento.quantidadeFr,
        atendimento.quantidadeFm,
        atendimento.setor,
        atendimento.leito,
        atendimento.convenioId,
        atendimento.dataEntrada,
        atendimento.dataSaida,
        atendimento.mesEntrada
      )
    );
  }

  return dtosAtendimento;
 } catch (error: any) {
  throw new Error(`Desculpe, um erro no servidor ocorreu! Erro: ${error}`);
 }
}
```

Pontos de melhoria

Método com muitos parâmetros: O método *getAtendimentos* recebe muitos parâmetros, seria melhor se o parâmetro esperado fosse um único objeto, onde os atuais parâmetros seriam campos desse objeto.

Método muito longo: O método *getAtendimentos* é muito longo, de modo a que boa parte de sua lógica poderia ser abstraída em diferentes métodos privados.

Adicionar comentários de *TODO*: Melhorias para esta classe estavam previstas, porém nenhuma destas está presente no código em formato de comentários *TODO*.

Falta de testes: A classe *AtendimentoRepository* não possui nenhum tipo de teste de *software*, nem mesmo unitários.

***DRY* (*Don't Repeat Yourself*)**: Ambos métodos possuem o exato mesmo trecho de código para montar o *dto* (*Data Transfer Object*) de atendimentos, quando este trecho poderia ser abstraído para um método auxiliar (*helper*).

***KISS* (*Keep It Simple Stupid*)**: No método *getAtendimentos*, os filtros por data de entrada e por ano utilizam o mesmo campo *dataEntrada* no banco de dados, de modo que caso o método receba os dois filtros como argumentos, o filtro implementado por último (por ano) irá sobrescrever o filtro anterior.

***SRP* (*Single Responsibility Principle*)**: A classe *AtendimentoRepository* está fazendo muito mais ações do que apenas realizar o *CRUD* no banco de dados, envolvendo e implementando diversas regras de negócio.

Pontos de destaque

Nomes de variáveis: Os nomes das variáveis e constantes estão bem descritivos e, na maioria dos casos, descrevem claramente seu propósito no código. Constantes como `PAGE_SIZE` seguem boas práticas e deixam evidente sua função. Variáveis como `skip`, `setorEnum`, `statusId` e `dataEntrada` são exemplos de nomes claros e semânticos, que facilitam a leitura e manutenção do código.

***SRP* (*Single Responsibility Principle*)**: A classe AtendimentoRepository tem muito mais ações do que apenas a criação do CRUD no banco de dados, seria melhor criar outras classes e atribuir as devidas responsabilidades a estas, suprindo melhor assim o SRP.

***DIP* (*Dependency Inversion Principle*)**: A classe AtendimentoRepository está implementando uma interface, o que traz organização e padrão ao código. Além de facilitar a criação de testes caso a classe sofra a injeção de dependências.

Formatação: O código está bem estruturado e segue boas práticas de formatação. O espaçamento está adequado, o que facilita a leitura e compreensão. A indentação é consistente, o que ajuda a manter a hierarquia clara entre os blocos de código. Além disso, o uso de espaçamentos entre funções e dentro de blocos de código permite um fluxo visual organizado. A formatação desse jeito contribui para a manutenção do código e facilita o trabalho em equipe, pois é mais fácil entender e modificar o que foi escrito. E segue a prática comum de typeScript utilizando a indentação de dois espaços

Conclusões

Apesar de possuir alguns pontos de destaque, é nítido que a classe *AtendimentoRepository* apresenta mais problemas do que qualidades, sendo muito bem-vinda uma refatoração completa, especialmente no método inicial, *getAtendimentos*, onde o nível de abstração é muito baixo, enquanto a quantidade de regras de negócio é muito alta.